Live Migration of an Entire Network (and its Hosts)

Eric Keller
University of Pennsylvania

Dushyant Arora, Diego Perez Botero, Jennifer Rexford
Princeton University

ABSTRACT

By decoupling applications from the underlying infrastructure, virtualization enables more flexible sharing of computing and network resources. Live virtual machine (VM) migration can move applications from one location to another without a disruption in service. However, applications often consist of multiple VMs and rely on the state of the underlying network for basic reachability, access control, and QoS functionality. Rather than migrating an individual VM, we show how to migrate an *ensemble*—the VMs, the network, and the management system—to a different physical infrastructure. Our LIME (LIve Migration of Ensembles) design leverages recent advances in Software Defined Networking (SDN) for a clear separation between the controller and the data-plane state in the switches. Transparent to the application running on the controller, LIME clones the data-plane state to a new set of switches. LIME then migrates the VMs, with both networks delivering traffic and maintaining synchronized state during the transition. Experiments with our prototype, built on the NOX OpenFlow controller, demonstrate the effectiveness of live migration of entire networks.

1. INTRODUCTION

Virtual machines (VMs) have emerged as a key technology for sharing resources, and seamlessly migrating running software from one location to another. However, a VM rarely acts alone. A VM often has dependencies with other VMs (e.g., as part of a multi-tier web application) and the network (e.g., to reach the addresses of other VMs, perform access control, and enforce QoS policies). Rather than migrating individual VMs, we show how to migrate an entire *ensemble*—the VMs, the network elements, and the management system—while the hosts continue communicating seamlessly across the transition.

1.1 Case for Migrating an Entire Network

Live VM migration gives network and service administrators the flexibility they need to consolidate servers, balance load, perform planned maintenance, and optimize user performance, without disrupting the applications. In addition, *wide-area* migration between different locations is gaining traction as a way to enable disaster avoidance, data-center consolidation and expansion, and load balancing across sites [10, 3]. Yet, today's multi-tier applications often consist of multiple VMs, with significant interaction between VMs in neighboring tiers. Placing these VMs near each other improves performance and reduces network overhead. For these applications, migrating a single VM in isolation could lead to significant performance degradation and high bandwidth costs to "backhaul" traffic to the other VMs. As such, migrating a single application may require joint migration of a group of related VMs [4, 11].

These applications are often tightly coupled with the underlying network. The network provides reachability between the application VMs, directs traffic through virtual appliances like load balancers and firewalls, and applies resource-allocation policies like routing and packet scheduling. Though early cloud offerings give tenants a relatively simplified view of the network, customers increasingly demand more sophisticated network functionality. In fact, cloud providers can offer each "tenant" a virtual network with the topology and configuration customized to the needs of the application [28, 6]. With the growing dependence on the network configuration, we believe the network (and network-management services) should migrate along with the collection of virtual machines. The ability to migrate an entire network ensemble would enable important new capabilities:

Migration within private infrastructures: Within a private infrastructure, migration would simplify the reallocation of resources within a data center, as well as the merging of server and network resources after an acquisition. One organization's VMs and network elements could migrate to run on top of the physical infrastructure of the other organization, without requiring any configuration changes or service disruptions. Migration is also useful for disaster preparation, such as evacuating a running data center in advance of a hurricane. Similarly, a cloud provider could migrate a nano data center from one container to another to improve reliability and performance, or reduce cost.

Migration to/from/between public clouds: Live

migration can simplify the transition to and from a public cloud. An enterprise could move its applications "to the cloud" without a transient disruption in service. Alternatively, a company could launch a new service in the cloud, and then move service to its (cheaper) private infrastructure once the workload becomes predictable [31]. Migration can also enable seamless changes between data centers, or even cloud providers. A customer may move to a new cloud provider for better performance or lower cost, or a cloud provider may (temporarily) move some tenants to another location another provider, during a period of high demand.

Moving target defense: In a public cloud, a tenant is vulnerable to attacks where adversaries (including other tenants) learn what servers and switches provide a service [24]. This enables both side-channel attacks (where the adversary infers sensitive information through resource usage patterns) and denial-of-service attacks (where the adversary sends targeted traffic that exhausts system resources). To thwart potential adversaries, the cloud provider could periodically change the placement of the tenant's VMs and virtual network components, all while presenting the abstraction of a stable topology and configuration to the tenant.

In each case, live migration of an ensemble minimizes performance disruptions and completely avoids the unwieldy, error-prone task of reconfiguring a complex system of servers and network elements in the new location.

1.2 Migrating a Software Defined Network

While major server virtualization platforms already support live VM migration, the migration of *network* state has received relatively little attention. A notable exception is the VROOM system [27] that migrates the control and data planes of a single virtual router, transparent to the end hosts and neighboring routers. However, VROOM does not naturally support other network functionality (such as switches, firewalls, and load balancers) and cannot simultaneously migrate a collection of hosts and network elements. In addition, VROOM understandably requires changes to the router implementation, impeding adoption. In contrast, our goal is to migrate an entire ensemble without *any* modifications to the server and network equipment.

To support network migration in a generic way, we separate the control-plane state and logic from the individual network elements by capitalizing on the recent trend of Software Defined Networking (SDN) [2, 19, 1]. In SDN, a logically-centralized controller runs management applications that directly control the packet-handling functionality in the underlying switches. For example, the OpenFlow API allows the controller to install rules, query traffic counters, learn about topology changes, and process data packets [20, 2]. Each rule matches bits in the packet header and performs simple actions (e.g., drop, forward, flood, rewrite, or send to the controller) on the matching packets. Priorities disambiguate between overlapping rules, and hard and soft timeouts allow switches to discard rules automatically after a fixed time interval or a period of inactivity.

Over the last few years, researchers have used OpenFlow to support a wide variety of network functionality, from conventional IP routers and learning switches to dynamic access control, server load balancing, seamless VM migration, and energy-efficient networking [9, 22, 14, 26, 12, 15, 21]. Numerous switch vendors support the OpenFlow API, and several commercial and open-source controller platforms are available. Several companies, university campuses, and research backbones have deployed OpenFlow switches.

1.3 LIME: LIve Migration of Ensembles

SDN, and the OpenFlow API, gives us a way to perform network migration without tying our solution to specific control-plane protocols or network functionality. While SDN makes a generic network-migration solution possible, many difficult challenges remain:

- The controller may run *arbitrary* applications that must continue to operate correctly during and after the migration. As such, our solution cannot exploit knowledge of the controller application to simplify or expedite the migration process.
- The network may have a large amount of state spread across the controller, the application VMs, and a distributed collection of switches. As such, our solution cannot afford to stop all communication between VMs while migrating this state.
- To minimize disruption, live VM migration relies on iterative copying of state while the VM continues executing on the old host, followed by a brief "freeze" of the VM during the final state transfer before restarting on the new host. However, even a temporary freeze of the entire network (or a single switch) would be too disruptive. Instead, our solutions must allow *both* the old and new networks to carry data traffic during the transition.
- The network state changes over time, through events sent to the controller and commands sent to the switches. As such, our solution must maintain the consistency of this state (e.g., the switch rules and traffic counters) while migrating to a new network.
- The application VMs may exchange a large amount of traffic and suffer performance degradation due to congestion or high round-trip times. As such, our solution must coordinate the migration of VMs and the switches to minimize "backhaul" traffic between the old and new networks.

Our LIME (LIve Migration of Ensembles) architecture addresses these five challenges.

LIME performs live migration of an entire network of VMs and OpenFlow switches. LIME runs on the SDN controller, underneath the management application, to provide the application with the illusion of a single topology while gradually moving all of the components to a new location. LIME allows both the old and new network to operate during the transition, by synchronizing the switch state and creating tunnels to relay traffic between the two networks. Our migration algorithms are carefully designed to avoid correctness problems that could arise from differences in the orderings of data packets or control messages in the two networks. LIME can also schedule the migration of VMs based on how much network traffic they exchange, to minimize the backhaul traffic during the transition. Experiments with our prototype system, built on top of the NOX OpenFlow controller, demonstrate that our migration algorithms perform well in practice.

The remainder of this paper is organized as follows. Section 2 discusses the many kinds of state in the network, and the consistency requirements they impose on the migration process. In Section 3, we present an overview of the LIME architecture, and motivate operating both the old and new networks during the transition. Section 4 introduces our algorithms for migrating VMs, switch state, and data traffic to the new network elements. We present our prototype implementation in Section 5, followed by the performance evaluation in Section 6. Section 7 compares LIME to related work, and Section 8 concludes the paper.

2. NETWORK STATE CONSISTENCY

To perform live migration in a software defined network, we must identify the many sources of network state. In this section, we first discuss the state in the application VMs, the controller application, and the network elements. Since we can apply existing migration techniques to the application and controller software, the remainder of the section focuses primarily on the consistency requirements for the various kinds of state in OpenFlow 1.1 switches.

2.1 Ensemble State

Migrating an ensemble involves transferring the state for three different kinds of components:

Virtual machines: Virtual machine migration technology is fairly mature. To migrate the state of the application and operating system, the hypervisor first copies any static state (such as files) and then iteratively copies the memory. In the meantime, the VM continues to run. Then, the hypervisor temporarily "freezes" the VM for the final state transfer, typically lasting just a few tens of milliseconds, and restarts the VM in the new location. In LIME, we leverage existing techniques for migrating individual VMs. In practice, a group of related VMs may have significant overlap in their state, including common operating-system images. In the future, we plan to exploit redundancy in the state of multiple VMs to minimize the migration overhead.

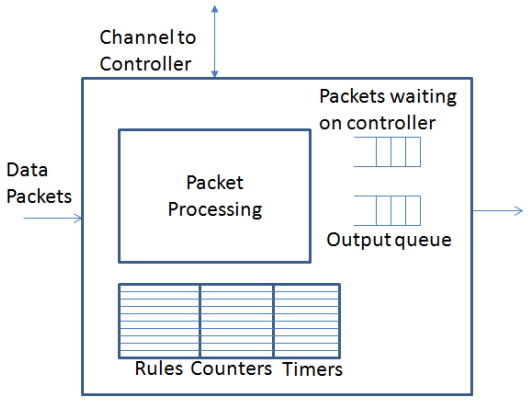

Figure 1: Switch state.

Controller application: In a software defined network, the logically-centralized controller runs a controller application that interacts with the underlying switches using a standard API like OpenFlow. This application may perform arbitrary computation and maintain arbitrary state. Fortunately, we can leverage standard VM migration techniques to move the controller application from one physical server to another.

OpenFlow switches: The switches forward data packets by applying rules installed by the controller. An OpenFlow switch does not encapsulate its state in a virtual machine that can be easily copied from one location to another. The switch state includes the flow tables, which contain the packet-handling rules, byte and packet counters, and timers for expiring the rules, as shown in Figure 1. In addition, the switch may have a queue of packets buffered awaiting further instructions from the controller, as well as control messages exchanged with the controller in both directions. The switch also has buffers for regular data packets queued for transmission. Next, we consider the consistency requirements for this state.

2.2 Switch Consistency Requirements

Controller applications must continue to operate correctly during and after the migration of an entire ensemble. Even in the absence of migration, these applications cannot assume they run on a perfectly reliable network. The network may drop or reorder data packets, and control messages sent to/from different switches may arrive out of order. Our migration algorithms can exploit these characteristics. In addition, we can leverage some useful features in the OpenFlow protocol to

simplify the migration process. Still, our algorithms need to respect whatever consistency properties are ensured by OpenFlow 1.1 switches in the absence of migration.

2.2.1 Flow Table

The first set of requirements come from the flow table, which consists of rules, traffic counters, and timers.

Rules: The controller installs rules in the flow table, where each rule consists of a pattern, a set of actions, timeout parameters, and a priority. When migrating an ensemble, the new switch should contain the same set of rules as the old switch before starting to handle data packets. After receiving the entire snapshot, the new switch may add and delete rules at the behest of the controller. OpenFlow messages are delivered over a TCP connection that ensures reliable, in-order delivery. However, the controller application cannot make assumptions about the delay for a switch to receive or apply these commands.

Traffic counters: The flow table maintains byte and packet counters for the traffic matching each rule. While these counter values do not directly affect the behavior of the switches, the controller application can query these counters and, based on the values, adapt the rules installed in the network. As such, we must preserve the counter values seen by the controller application across the migration. However, the OpenFlow API does not offer a way to set the initial counter values in the flow table. Fortunately, counters are cumulative, making it relatively easy to combine values collected from the old and new switches to present a consistent view to the application.

Timers: The flow table also maintains timers for expiring rules. A rule may have a hard timeout (to delete the rule after a fixed time has elapsed) and a soft timeout (to delete the rule if no packets arrive for some period of time) The OpenFlow API does not expose the current values of the timers, making it difficult to migrate this state. Yet, the specification also does not offer hard guarantees on timer accuracy, or exactly when the switch installs a rule (and, hence, starts the timer). As such, we argue that network migration can use a relaxed notion of time accuracy, as discussed later in Section 4.

2.2.2 Control Traffic

The second set of requirements concern the control messages sent between the switch and the controller.

Control channel: An OpenFlow switch communicates with the controller over a TCP connection. The controller sends commands to (un)install rules, query traffic counters, and send packets, and the switch sends events such as links failing/recovering, the values of traffic counters, and packets requiring further handling. While the TCP connection ensures these messages arrive in order, the messages may experience arbitrary delays. A switch may not apply a command immediately. Fortunately, the controller can issue a "barrier" command to verify that the switch has applied all previous commands[1]. While the TCP connection ensures that control messages arrive in order, the controller application cannot assume that control messages sent to/from *different* switches arrive in order.

Switch and port status: A switch maintains information about its incident links (or "ports"). If a port goes up or down, the switch sends a control message to the controller. As such, the controller application learns about topology changes, albeit with a delay. When a port fails, the switch drops packets directed to that port. When a switch fails, the controller loses the control channel to the switch, and regains the connection on recovery. As such, the controller relays switch join/leave events to the application. The application may respond to topology changes by changing the rules on one or more switches. While the network must present an accurate view of the topology to the application, delays in learning about topology changes are inevitable, even in the absence of migration.

2.2.3 Data Packets

The final set of requirements concerns data packets traveling through the two networks during the migration process.

Packets buffered awaiting controller instructions: When traffic matches a rule with a "send to the controller" action, the switch sends a message (containing the packet header) to the controller and buffers the entire packet awaiting further instruction. This queue of "packets in waiting" is part of the state of the switch. Upon receiving and processing a "packet in" event, the controller application may instruct the switch to handle this packet. However, packet loss is acceptable in a best-effort network, even in the absence of migration. A packet could be dropped without triggering a control message, or dropped after the switch follows the controller's instructions on how to handle the packet. This offers some flexibility to decline to handle some packets sent to the controller in the midst of a migration, as long as "packets in waiting" are not buffered indefinitely.

Packets queued for transmission: The switch also has a buffer that stores packets ready for transmission. Under normal circumstances, these packets may be dropped on the outgoing link due to congestion, cor-

[1]A recent study shows that many of today's OpenFlow switches do not implement the barrier command correctly [25]. For the purposes of this paper, we assume the switches correctly obey the OpenFlow 1.1 specification. Ensuring the controller application operates correctly in the face of buggy switches is extremely difficult, even in the absence of migration.

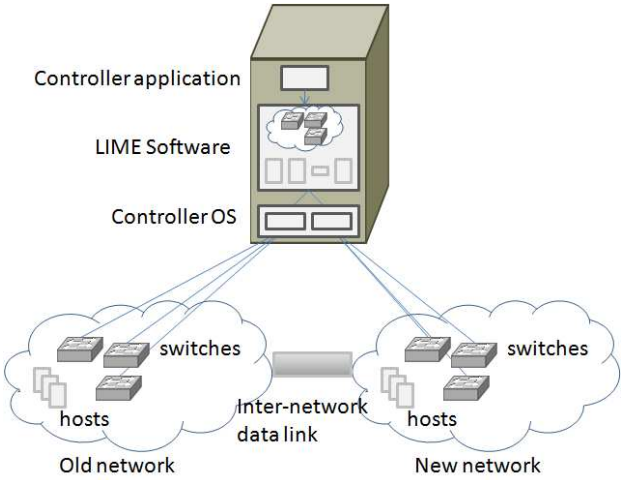

Figure 2: Overall architecture.

ruption, or other reasons. The controller application cannot assume that data packets are delivered successfully, and does not receive information about the specific causes of packet loss. While packet loss does affect performance, it does not affect correctness. As such, dropping buffered data packets during the migration process would not compromise the correct operation of the controller application. If these dropped data packets are themselves part of a control message sent to or from the controller, the TCP connection between the two components would retransmit the data, ensuring reliable delivery of the control messages.

3. LIME ARCHITECTURE

Live migration means that the applications running on the end hosts continue to operate without disruption. Today's VM migration solutions can limit the downtime to tens of milliseconds, or at most a few seconds, depending on the workload [8]. Live migration of the entire network means that the VMs not only continue processing, but also communicating with each other under the same network configuration.

Figure 2 show our overall architecture. The hosts and switches migrate from an *old* physical network to a *new* physical network. To perform migration and handle temporary data traffic between the old and new networks, the two networks must have some network connectivity, such as a a dedicated physical link or a tunnel over the Internet. The controller could run in either physical network, or be migrated from one to the other separately from the migration of the hosts and switches. On the controller, the LIME software acts as a transparent proxy between the controller operating system (e.g., NOX [13]) and unmodified controller applications. The LIME software has a view of both the old and new networks, but presents a view of a single network to the controller application.

In this section, we first describe a strawman migration solution where only one of the two networks (or switches) handles data traffic and interacts with the controller at a time. However, this solution is slow and inefficient. We then discuss how LIME allows both networks to handle traffic simultaneously during the transition, leading to an efficient, live migration solution.

3.1 Stop-and-Copy Migration is Not Efficient

Given the success of live VM migration, it may seem natural to use a similar mechanism for network migration. Live VM migration uses an iterative copying process to minimize downtime. First, an initial snapshot of the VM state is copied to the new physical server. Since transferring the state may take several seconds or even minutes, the VM continues running on the old server, while the virtualization software tracks which pages in memory have been dirtied. A snapshot of these dirty pages is copied during the next iteration. Eventually, this snapshot is small enough, and the VM is stopped, the last bit of state is copied over, and the VM resumed at the new location. Migration is considered *live* because the period when the VM is frozen is imperceptible to applications.

Applying the same approach to network migration is not efficient enough, even with the clear separation between the controller and the switches. In fact, even applying stop-and-copy migration to individual switches (rather than the entire network) is not efficient, as it results in excessive "backhaul" traffic between the old and new networks.

3.1.1 *Migrating the Network is Not Live or Efficient*

Inspired by live VM migration, we could perform network migration in three main stages: (i) iteratively copying the host and switch state from the old network, (ii) freezing the old network for the final state transfer, and (iii) starting the new network. In this solution, only one physical network would handle data traffic and interact with the controller at a time. However, freezing all of the VMs for a final copy of all of the host and network state would lead to long delays. Instead, the solution should allow individual VMs to start running in the new location, while the rest of the state transfer continues. To allow the old network to handle all of the traffic, the new network could simply direct the VM-to-VM traffic through the old network using tunnels. For example, a VM running in the network could send and receive traffic via a tunnel that terminates at its old location in the other network, to allow the old switch to handle all of the traffic. These tunnels can be implemented using existing OpenFlow mechanisms for pushing and popping headers (e.g., VLAN) on packets.

Unfortunately, this strawman approach has several limitations. First, the solution leads to excessive back-

haul traffic. For example, all traffic between VMs in the new network would travel back and forth through the old network—traversing the inter-network pipe twice! both run in the new location must In fact, right before the final state transfer to the new network, *all* traffic between VMs would travel through the old network. Second, unlike individual VMs, we cannot restart the network in the new location in a single step. To start using the new network, the controller must simultaneously remove all of the tunnels directing traffic through the old network. This is not possible in a large, distributed collection of switches. Alternatively, the controller could cut all communication via the inter-network link before starting to remove the tunnels. However, this would lead to a large burst of packet loss, significantly degrading performance, and cannot be considered "live" migration.

3.1.2 Migrating Individual Switches is Not Efficient

Rather than migrating the entire network in a single step, we could apply an iterative migration process to each individual switch. That is, the controller could copy the switch state from one physical switch to another, temporarily freeze for the final state transfer, and then start the switch in the new location. Compared to migrating the entire network, migrating an individual switch involves less state and simplifies the "instantaneous" cut-over to the new physical location. In this approach, some switches may run in the new network, while others continue running in the old, but each switch runs in one place at a time. The controller can ensure that data traffic flows during the transition by installing tunnels between "neighboring" switches that temporarily reside in different networks, These tunnels ensure that packets traverse a sequence of active switches with the appropriate rules installed.

To migrate an individual switch, the controller first installs tunnels to and from the new physical switch, to prepare for the transfer. Then, once the state transfer is complete, the controller can (i) install tunnels in the old switch (to direct traffic to the new switch) and (ii) remove the tunnels in the new switch (to start handling packets directly). Migrating the individual VMs is similar to switch migration. As in the the network-wide migration solution, tunnels can direct traffic to/from a VM in the new network via an incident switch still running in the old network. Once both the VM and the incident switch have migrated, the VM can communicate directly through the physical switch.

Since the stop-and-copy phase is nearly instantaneous, migration is "live." Unfortunately, this approach can be very inefficient. In the worst case, a packet may traverse a tunnel between the two networks at every hop in its journey. Even if both the sending and receiving VMs reside in the new network, the path between them

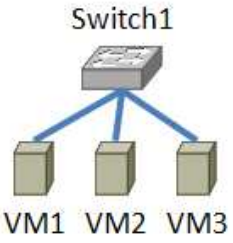

Figure 3: Example to illustrate migrating switches leads results in VM communication requiring at least one round-trip between networks even when two VMs are located in the same network.

may traverse switches still running in the old network, requiring multiple traversals across the tunnels. Consider the simple tree topology in Figure 3. If we migrate Switch1 first, then any communication between the VMs would travel back and forth through the new network. If instead we migrate VM1 first and then VM2, the traffic between them would travel back and forth through the old network to traverse Switch1. If instead we migrate VM1 and Switch 1 first, traffic between VM2 and VM3 (both running in the old network) would travel through the new network. Even with this simple topology, the packet latency and network overhead is significant during the transition.

3.2 Cloning the Network is Live and Efficient

Rather than migrate the network with a stop-and-copy phase, we instead *clone* the network state and allow both the old and new networks to handle traffic and interact with the controller simultaneously. During this time, two physical switches simultaneously act as the single logical switch they represent.

3.2.1 LIME Components

To clone the network, we need to (i) maintain consistency between the two networks, (ii) preserve the transparency to the controller application, and (iii) coordinate the actual migration process. In the LIME software architecture, shown in Figure 4, three main components are responsible for these three important aspects of the live migration.

Synchronize flow table: When the controller application issues commands, the controller must update both networks. The *synchronize flow table* component installs flow-table entries in both networks and keeps the state consistent. Similarly, this module is responsible for any transformations of the rules necessary to allow the two physical switches to operate concurrently, as discussed in further detail in Section 4.

Preserve network view: In the reverse direction, the controller must present the controller application with a

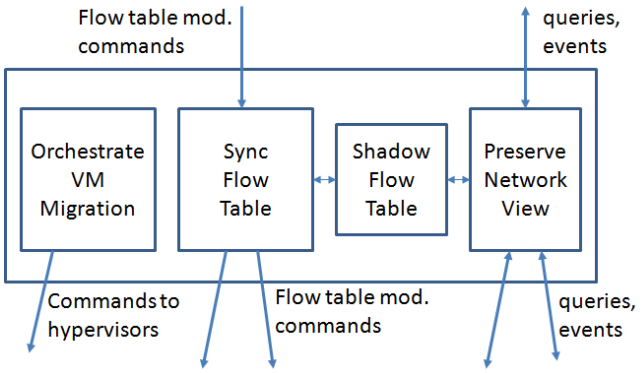

Figure 4: Architecture of LIME Software.

consistent view of a single underlying network, despite streams of events coming from the two sets of switches. The *preserve network view* component is responsible for handling packet-in events, unplanned topology changes, and traffic statistics arriving from the two networks during the transition.

Orchestrate VM migration: The performance of live network migration depends on the communication patterns between the VMs. Rather than migrating all VMs at the same time, the controller can compute an efficient schedule for migrating the individual VMs to minimize the amount of inter-network traffic. In addition, the *orchestrate VM migration* component issues the commands necessary to migrate the individual VMs.

In our current design and prototype, we focus mainly on the first two modules, and use a relatively simple algorithm to orchestrate the VM migration.

3.2.2 *Ensemble Migration Process*

In order to perform live migration, we first clone the switch state to the switches in the new topology. We then migrate VMs individually with both the new and old network actively handling traffic, while the LIME software maintains the consistency of the state in the two networks. Once all VMs are migrated, we finalize the process by synchronizing any remaining state differences and begin using the new network exclusively. The controller itself can migrate at any time (e.g., as a final step in the process).

Clone switch state: As the first step migrating to a new network, we must first synchronize all of the switch state so the new network operates the same way as the old network. Since the switch state changes constantly, we cannot instantaneously install all flow table entries in all switches. Similar to live VM migration, we apply an iterative copying process. To get the state, we could poll each of the switches to obtain the current flow table. However, doing so will take time and potentially result in an inconsistent snapshot due to not being able to query all of the switches at the same time. Even more, it is not necessary. We can capitalize on the position of LIME in between the controller application and the switches and maintain a *shadow flow table* in the LIME software. We obtain the first set of state to install in the new network by taking a snapshot of the shadow flow table. While that state is being installed in switches in the new network, we allow the controller application to continue issuing commands, which are immediately sent to the switches in the old network. We keep track of those commands (insert and delete rules) and, when the installation of the flow table for the previous iteration completes, we then create another snapshot with only the commands since the last snapshot. Since snapshot is typically smaller, allowing installation to complete more quickly, making the next snapshot even smaller.

This process continues until the snapshot is small enough to effectively be installed instantaneously. Then, we block the controller application from installing any new rules (by buffering up the commands from the controller application but not acting on them). In the meantime, data packets continue flowing through the old network. At this point, the cloning has completed and we simply need to keep the network state consistent as we start migrating the VMs and their traffic.

Migrate VMs with both networks active: At the end of the iterative switch state cloning, both networks can handle traffic and therefore we can start migrating the VMs. We assume that we cannot migrate all VMs concurrently due to bandwidth limitations—while disk images are likely shared among several VMs and therefore can make use of redundancy elimination, the in-memory working set is unique and likely represents several gigabytes per VM. To cope with this, we make use of tunnels between the two networks to maintain communication between all VMs. Also, we capitalize on the fact that both networks will have the same state to efficiently perform a live network migration. We discuss how we deal with the simultaneous use of two networks, while preserving the view of one network to the controller application in Section 4.

Copy traffic counters and timers: Once all VMs are migrated we finalize the process by copying over the remaining state that is not maintained by LIME—in particular, the traffic counters and timer values. The counters are read from each of the switches and stored in the shadow flow table. The timers are handled in a different manner which we discuss further in Section 4.

4. DEALING WITH TWO NETWORKS

To perform live migration of an ensemble, we allow both the old and new networks to handle traffic simultaneously. In this section we describe our algorithm for using tunnels to maintain *data-plane* connectivity during the migration. We then detail how we present the abstraction of a single *control plane* to the controller ap-

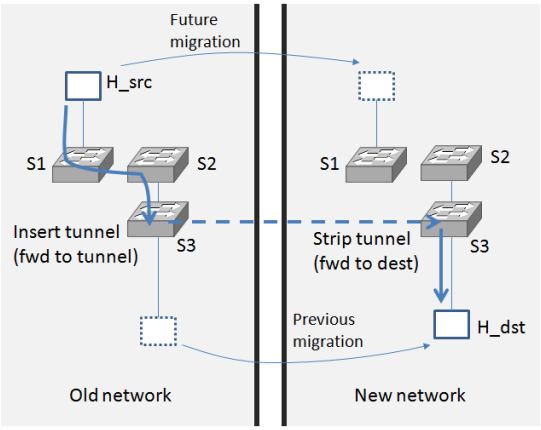

Figure 5: Example of tunneling at egress only.

plication, even though each logical switch corresponds to two physical switches during the transition.

4.1 Tunneling Between the Two Networks

By cloning the switches, and using both the old and new switches simultaneously, we can ensure that traffic only flows between the two networks when an already-migrated VM communicates with another VM that has not migrated yet. No traffic ever traverses the inter-network link more than once, let alone multiple times. For example, in Figure 5, host *H_src* in the old network on the left communicates with host *H_dst* in the new network on the right. By handling all traffic in the network of the sending VM we ensure that traffic only traverses the inter-network link when the destination VM resides in a different network than the sender.

We call our algorithm *tunnel at egress*. The link connecting the destination VM to the network is realized as a tunnel between the two physical switches that together represent the incident logical switch (e.g., S3 in the figure). The tunnel serves only to deliver the packet to a VM in a remote network and does not contribute to any timers or traffic counters. Response traffic from *H_dst* would traverse S3 and S2 in the new network, with switch *S1* in the new network forwarding the packets to the corresponding switch in the old network for delivery to *H_src*. Once *H_src* migrates to the new network, traffic between *H_src* to *H_dst* flows exclusively in the new network, without any tunneling.

To perform the tunneling, we must first calculate and install a path between each of the corresponding switches (e.g., for host *H_dst* this would be from switch *S3* in the old network to switch *S3* in the new network). Each VM in the network is assigned a tunnel identifier at the beginning of the process. We then install rules into the flow table of each forward the packet along the pre-installed path, based on the tunnel identifier. Once the tunnels are established, we modify the rules in the switches to use the tunnels by encapsulating the decapsulating the packets at the sending and receiving ends, respectively. We iterate through all flow-table entries and modify any rule whose action is to forward out of a port that is to be connected to a VM to instead (i) insert a tunnel header with the correct tunnel identifier, and (ii) forward out of the port that was determined as part of the switch to switch tunnel path calculation.

Once this process completes, we can migrate the VM. For each VM we need to direct traffic sent by VMs remaining in the old network. In the old network, we install the egress-tunneling rule at the old physical switch, and remove egress-tunneling rules at the incident switch in the new network. Removing these rules simply consists of re-installing the original rules that forward traffic directly to the destination VM.

4.2 Transparency to Controller Application

While simultaneously using switches in both networks, we must present a consistent view of a single network to the controller application. This requires careful handling of the switch state discussed earlier in Section 2.2.

4.2.1 Flow Table

The first set of requirements come from the flow table, which consists of rules, traffic counters, and timers.

Rules: LIME intercepts all commands from the controller application to add or delete rules in the flow table. The *Sync Flow Table* component in Figure 2 keeps the state in both networks synchronized by updating the shadow flow table and installing the rule to the corresponding switch in each network. For rules which require translation to instead insert a tunnel header, the *Sync Flow Table* module performs this translation.

Traffic counters: When querying a switch for particular counters, the controller application must receive an accurate response. Since traffic flows through both networks, the counters in the corresponding switches in both networks must be combined. Even more, any counters on switches from prior migrations must also be included. For this, as part of the finalization step, we read in the final statistics of the old network and store them in the shadow page table. This does not necessarily mean we have a growing amount of state, as the statistics are deleted as rules are removed from the flow table. Upon receiving a query from the controller application, the *Preserve Network View* component queries the corresponding switch in each network, and returns the summation of the two values (plus the value in the shadow flow table) to the controller application.

Timers: Timers are the most difficult class of state. Preserving the network view accurately means that the timers in the two switches should expire at the same time, and trigger a single notification to the controller application. Hard timeouts (which expire after a fixed time interval) are relatively easy to handle. LIME can

either install the rule with the timeout at the same time in both switches (for a new rule installed during migration) or install the rule with a reduced timeout (when cloning an existing rule). Further, the rule can instruct the switch to send a notification when the timer expires. Since the timeout occurs in both switches, the *Preserve Network View* modules receives both notifications but only delivers one to the controller application. For rules configured *not* to generate a expiry notification, the *Sync Flow Table* component modifies the rule to notify the controller (so the rule can be removed from the shadow flow table), without relaying the event to the controller application (to preserve the network view).

Soft timeouts (which expire after a period of inactivity) are more difficult to handle. Traffic flowing through either network may match the same rule, and the spacing of these packets may trigger timeouts in the two physical switches that would not occur if all traffic went through a single switch. In the example in Figure 6, suppose the idle timeout is set to 1 second. Packets pkt0 and pkt2 arrive 1.8 seconds apart at the old switch, and pkt1 and pk3 also arrive 1.8 seconds apart at the new switch. Each would trigger an idle timeout on the individual physical switch. However, when considering the two physical switches as one logical switch, the packets arrive 0.9 seconds apart, which should *not* cause a soft timeout. To deal with this, we would need to install an idle timeout that notifies the controller of the timeout without removing the flow entry, so the controller could explicitly delete the rules if necessary. Unfortunately, the current OpenFlow specification does not support this feature. However, we can mimic coarse-grained timers by using "permanent" rules, and periodically polling the traffic counters to see if any traffic has matched the rule since the previous poll.

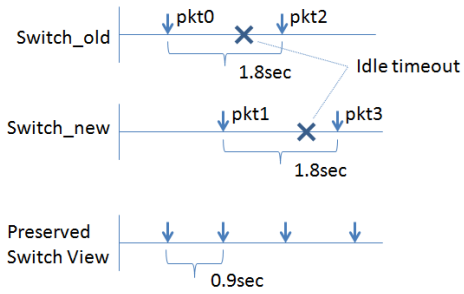

Figure 6: Example of idle timeout.

4.2.2 Control Traffic

The second set of requirements concern the control messages sent between the switch and the controller.

Control channel: OpenFlow messages sent to/from *different* switches over different TCP connections may arrive in any order, due to arbitrary delay and packet loss in the network. However, messages sent to/from a *single* switch, over a single TCP connection, should arrive in order. As such, LIME cannot simply combine control messages sent by the two physical switches (representing a single logical switch) in the order they reach the controller. In particular, if a packet traversing one switch triggers data traffic on the other, we cannot easily preserve the correct ordering of control messages at the controller.

Consider, for example, a stateful-firewall application that installs a rule that performs two actions on client traffic: (i) forwarding the traffic to the server and (ii) sending the packet header to the controller (e.g., to trigger installation of a rule permitting server traffic in the reverse direction). If the client-to-server traffic traverses one physical switch, and the resulting server-to-client traffic traverses the other, the response traffic may reach the second switch before the control message from the first switch reaches the controller! If a rule directs server-to-client traffic to the controller, the "packet-in" event for the server traffic may reach the controller application *before* the event for the client traffic—something that would not happen with a single switch.

An example is shown in Figure 7 where *H_src* sends a packet (1), the switch in the old network forwards the packet (2a) and sends the header to the controller (2b). After receiving the packet, *H_dst* sends a response (3). At the switch in the new network, since the controller has not installed a rule allowing this traffic, the packet is sent to the controller (4). With a single physical switch, with a single TCP connection, the controller application would receive the messages in order and know not to block the return traffic. However, if the forward and reverse traffic are handled by different physical switches (representing the same logical switch), the messages to the controller can be received in either order[2]. If the message due to the response packet (4) is received before the packet due to the forward direction packet (2b), the controller proceeds to install a rule to block all traffic for that flow—an undesirable outcome.

Fortunately, we can prevent these kinds of situations by modifying the rules installed in the flow tables. During the migration, we temporarily modify all rules that both forward traffic and send a message to the controller to instead only send to the controller. This inherently enforces a message ordering that respects the dependencies between events. As part of handling the control message, the LIME software simply instructs the switch to forward the packet, without informing the controller application. This gives the application the illusion of

[2]This holds even for when we migrate each switch individually rather than allowing two physical switches to simultaneously represent the same logical switch – as the message may be in flight when the migration occurs.

performing both packet-handling actions, while guaranteeing the correct message ordering during migration.

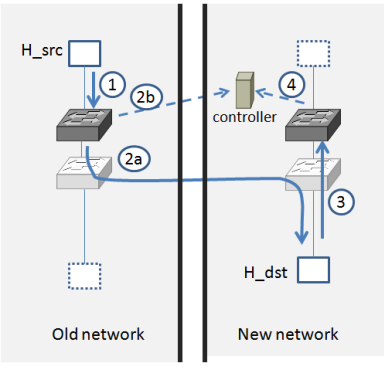

Figure 7: Example of problem of forward and send to controller.

Switch and port status: A topology change, such as the failure or recovery of a link or a switch, triggers a control message to the controller. To deal with these, we simply must ensure that the controller application is presented a consistent view of the network. That is, we present a topology that only includes switches and links that are alive in *both* networks during the migration process. When the migration process is complete, however, we present the status of the switches/ports in the new network to the controller application.

To do this, LIME intercepts these switch and port status messages. If the status message removes a switch or a link due to failure in one of the networks, LIME passes this event to the controller application, essentially making that switch/port unavailable in both networks. The failure may trigger the controller application to add, modify, or delete rules in the switches. As previously discussed, the *Sync Flow Table* component keeps both networks consistent.

Switch and port status messages can also indicate the recovery of a switch or port. If the switch or port is now live in both networks, LIME relays the recovery event to the controller application. Again, the *Sync Flow Table* handles any resulting modifications of the flow table. If, however, the new switch or port is only available in one of the two networks (e.g., the operator installed a new switch during migration), LIME adjusts its internal representation of the two networks, but does not pass the event to the controller application.

4.2.3 Data Packets

The final set of requirements concerns data packets traveling through the two networks during the migration process.

Packets buffered awaiting controller instructions: Packets that are sent to the controller require special attention. When an arriving packet matches a rule directing traffic to the controller, the switch places the packet in a buffer awaiting a response from the controller. Later, the controller application may issue a command to the switch referencing the buffer identifier for this packet. When (un)installing rules, the *Sync Flow Table* component must not instruct both physical switches to handle this packet. Instead, the module modifies the commands to tell the original switch to handle the packet, while simply installing rules in the other switch for handling future packets.

Packets queued for transmission: Data packets that match existing flow table entries and can be processed entirely in the switch do not cause any problems, in terms of preserving the correctness of the controller application. They can simply be queued for transmission at either switch.

5. LIME PROTOTYPE

We built an initial prototype of LIME to demonstrate live ensemble migration. We based our prototype on OpenFlow as the exemplar SDN technology. We tested with both a small testbed running Open vSwitch as the OpenFlow switches and VirtualBox as the virtualization technology supporting live migration, as well as with larger networks using the MiniNet emulator [17].

As shown in Figure 8, the prototype consists of the core migration components, along with a client console and server to control the migration process remotely. LIME was implemented as a modified NOX [13] library. This library exposes the same interface as an unmodified NOX controller, but intercepts event callbacks (e.g., *packet_in*, *flow_removed*) to preserve transparency to the controller applications. With this scheme, NOX applications communicate with LIME by importing the modified library. In turn, the LIME software communicates with the NOX core components, which provide the basic OpenFlow controller functions. These components then interact with OpenFlow switches. We implemented both the ideal algorithm (which uses both networks simultaneously) and the strawman *stop-and-copy* algorithm (which uses one network at a time).

In addition to providing an interface to the controller application, network operators need to control the migration process. The LIME prototype was designed with a client-server model to enable the processing of migration commands at any given time. However, as NOX follows event-based programming and does not support multithreading, we run an additional process to monitor migration-related commands and their execution.

The LIME client console then coordinates the network migration process by sending commands such as *PREPARE* which triggers LIME to start populating the new switches, and *MIGRATE_HOST* which triggers LIME to send commands to the specified server

(to migrate a VM) and to the OpenFlow switches (to establish tunnels).

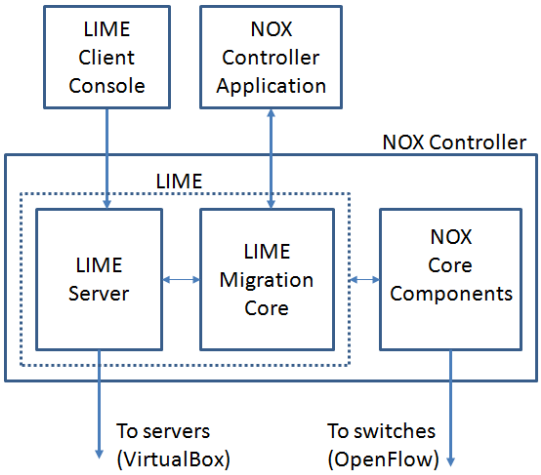

Figure 8: LIME Prototype.

6. PERFORMANCE EVALUATION

In this section we evaluate live ensemble migration with LIME using MiniNet. We first illustrate the difference between LIME, which uses both networks simultaneously, and the *stop-and-copy* strawman described in Section 3.1.1. We then evaluate the impact the order of VM migration can have on the overall performance. Finally, we evaluate the impact the bandwidth dedicated to VM migration has on the application performance.

6.1 LIME vs. Stop-and-copy

To illustrate the differences between LIME and Stop-and-Copy, we show a fine-grained view of the throughput between two VMs as well as a coarse-grained comparison with aggregate statistics.

We first performed a small-scale experiment where each physical network consists of two hosts and four switches, as shown in Figure 9. VM1 communicates with VM2 for the entire duration of the ensemble migration. We used a single switch connected to all switches to represent the inter-network data pipe, and use a simple mechanism for determining tunnel paths. We use *netem* to enable a configurable delay in this switch representing the inter-network data pipe delay. For VM migration with MiniNet we emulated a VM migration by connecting each host to an intermediate switch with connections to both networks. As such, mimicking migration is a simple matter of directing packets to a different output port connected to the new network.

The performance of the migration in terms of packet loss, throughput, and latency depend on the creation and deletion of tunnels, and not on the particular NOX controller application. LIME ensures consistency independent of the application, as discussed in Section 2. As

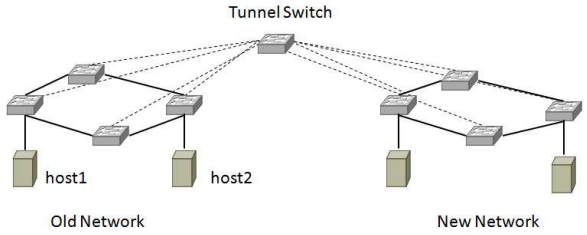

Figure 9: Small-scale network.

such, for performance measurements, we use a simple NOX controller application (pyswitch) which functions as an Ethernet learning switch. In each VM we ran D-ITG [7] to measure performance to the other VM.

We then performed a live ensemble migration with both LIME and the strawman *stop-and-copy*. Figure 10 shows the average delay between VM1 and VM2 for each 1-second time interval. VM1 migrates at time 40s. In both LIME and stop-and-copy, the delay increases since all traffic incurs the added one-way latency to traverse the tunnel. The corresponding throughput (not shown), drops due to the increased round-trip time (RTT). VM2 migrates at time 80s. In LIME, since both VM1 and VM2 are in the same network, the delay returns to the original delay (and the throughput recovers to the original value). In stop-and-copy, however, the delay further increases as packets have to make a round trip through the old network. At time 120s, we switch over to using the new network. In stop-and-copy, this involves cutting communication with the old network and then removing the ingress tunnels at all of the switches in the new network. For this short period, all packets are dropped. We artificially added 200ms of delay to make this perceptible—using a small topology and having all switches and the controller on the same server does not experience any delay that a real network would. This packet loss does not affect the average RTT, but there is a small spike in packet loss seen with our measurement.

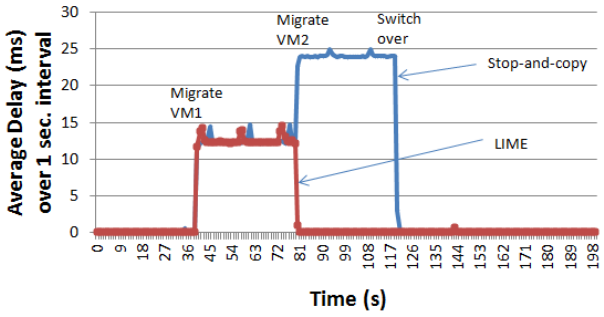

Figure 10: An illustration of the differences in efficiency between stop-and-copy and LIME.

To compare LIME and stop-and-copy at a larger scale,

we use a network consisting of 10 hosts and 10 switches. We used a traffic pattern, *k-random*, where each host sends traffic to k randomly chosen hosts. As was the case with the small network, having tunnels with very little delay diminishes the advantage LIME has in terms of efficiency—stop-and-copy still experiences packet loss due to the freeze when switching from the old network to the new network. To evaluate this effect, we varied the delay through the tunnels and measured the average delay across all packets. As shown in Figure 11, LIME is less affected by the inter-network tunnel latency.

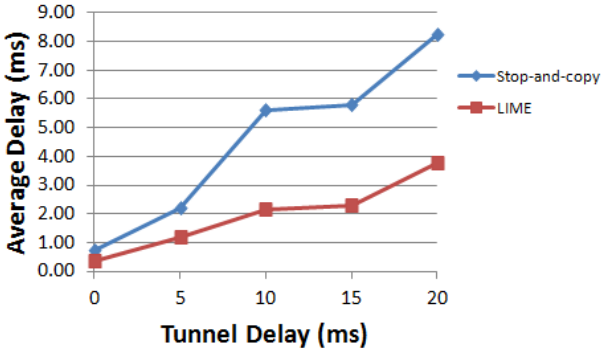

Figure 11: Impact on intern-network latency on the efficiency of LIME vs stop-and-copy.

6.2 Impact of VM Migration Ordering

With LIME, two communicating VMs in the same network do not experience extra latency of traversing a tunnel and inter-network link. As such, the order in which VMs are migrated impacts the performance of the applications running in the hosts, as well as the amount of traffic traversing the inter-network link. We experimented with a network of 10 hosts and 10 switches and varied the order of VM migration—performing one migration at a time every 5 seconds.

To evaluate the benefits of VM migration scheduling, we conduct a simple experiment that explores many different schedules. For our experiment, we again assume a k-random traffic pattern, with each VM sending traffic uniformly to k destinations. To determine the "good" and "bad" orderings, we generated 200 random possible and calculated the cost of the ordering—by counting inter-network traffic that would occur after each migration. We then collected the average delay over all traffic for each time interval during the entire ensemble migration. As can be seen in Figure 12, we can see that the bad ordering rises to a higher average delay sooner than the good ordering and the good ordering recovers to the original delay quicker than the bad ordering.

6.3 Impact of VM Migration Bandwidth

Live migration of a single VM consumes bandwidth.

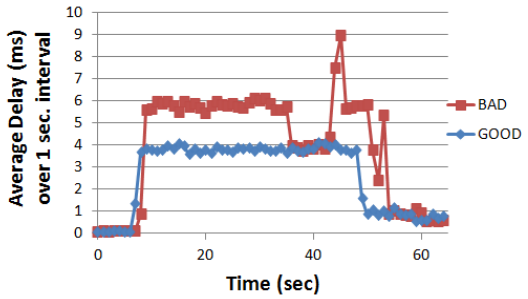

Figure 12: Impact of the ordering of VM migrations in LIME.

Yet, backhaul traffic between VMs in the old and new networks also consumes bandwidth. To understand the tension between these sources of overhead, we vary the number of VMs migrated at the same time. Intuitively, migrating as many VMs as possible makes the entire migration process faster, reducing the inter-network traffic due to inter-VM communication. However, as shown in Figure 13, migrating too many VMs at the same time does not strictly improve the performance in terms of total or average traffic rates. In particular, backhaul traffic increases if several VMs in the new network communicate heavily with VMs in the old network. This points to the need for smart algorithms for scheduling the final state transfer for VMs moving from one network to another. In future work, we plan to explore VM migration algorithms that identify the densest subgraph of communicating hosts [5], and migrate these VMs as a group to minimize backhaul traffic.

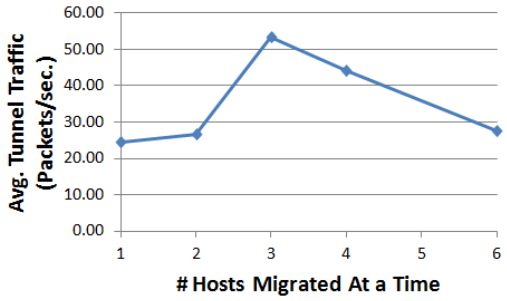

Figure 13: Impact on bandwidth available for migrating multiple VMs.

7. RELATED WORK

In this paper we motivate the need to migrate an entire network and its hosts. Different aspects share ideas with some prior research.

Migration of Network Elements: Perhaps the most closely related research is the work to migrate or clone network state. Migration has shown great use in

network management—VROOM demonstrated migration of a virtual router [27] while router grafting demonstrated migrating a single BGP session [16]. In contrast, we migrate an entire network and target software defined networks where the network is more stateful. However, we were inspired by the approach in VROOM to first setup the network before migrating end points. The VROOM approach was duplicated in [23] with a different virtualization technology, and they extended it to OpenFlow networks as well. However, they simply treated the switches as the forwarding tables (as in VROOM) and did not deal with the many issues related to consistency, correctness, and transparency to the controller application that LIME addresses.

Cloning of Network State: Cloning has shown great use in network debugging. Analogous to gdb for software programs, ndb enables the same type of debugging inspecting and controlled execution by cloning the network state at each of the routers and executing them in a parallel virtual network [18]. OFRewind, works in a software defined network to record data and control traffic to enable replay at some future time [30]. In order to not require a log of all messages since the beginning time of the network, they enable an optional snapshot of the flow tables, though do not address the other state as we discuss in this paper. In both ndb and OFRewind, however, at the point of cloning, the relation between the original network and the cloned network stops—in contrast to LIME where we perform a live migration.

Migration of VMs across networks: Sharing some motivational scenarios (such as disaster preparation and data center load balancing), others have looked into migrating between data centers over. VMWare and Cisco even provide a commercially available solution [10]. In general, this work has relied on the principle of extending the Layer 2 subnet across data centers. For example, CloudNet [29] uses Multi-Protocol Label Switching (MPLS) based VPNs to create the abstraction of a private network and address space shared by multiple data centers. In contrast, with LIME we are migrating both the VMs and the network and so are not limited to technologies that extends the local network.

Migrating multiple VMs: We noted that VMs do not typically operate in isolation, and so should be migrate as a collection. The efficiency of doing this has been studied. In one case, all VMs are co-located on the same server and being migrated together to a new server [11]. In this case, optimizations implemented in the hypervisor to capitalize on shared or similar pages in memory can reduce traffic. In another case, migrating multiple VMs (not necessarily on the same server) together can capitalize on the fact that they often will a great amount—e.g., running the same OS image [4]. These works are complementary to ours as we can capitalize on more efficient VM migration mechanisms.

8. CONCLUSIONS AND FUTURE WORK

Live VM migration and, more recently, live *wide-area* VM migration have become staples in the management of many data centers and enterprises. Yet, today's technology simply considers each VM independently. However, a VM rarely acts alone. A VM interacts with other VMs as part of a larger application and relies on the underlying network for communication. As such, rather than rather than migrating individual VMs, we show how to migrate an entire *ensemble*—the VMs, the network elements, and the management system.

Our LIME (LIve Migration of Ensembles) design leverages recent advances in Software Defined Networking (SDN) and (transparently to the application running on the controller) clones the data-plane state to a new set of switches. LIME then migrates the VMs, with both networks delivering traffic and maintaining synchronized state during the transition for an efficient and live ensemble migration. This migration is live since the hosts continue communicating seamlessly across the transition, and it is efficient since our "tunnel at egress" algorithm keeps traffic in one physical network whenever possible. Experiments with our prototype, built on the NOX OpenFlow controller, demonstrate the effectiveness of live migration of entire networks.

As future work, we plan to explore algorithms for scheduling VM migration. This includes determining the measurement needs and the timing of VM migration. This research includes leveraging technologies like redundancy elimination) to reduce the overhead of copying the state for multiple related VMs.

9. REFERENCES

[1] Open Networking Foundation. `https://www.opennetworking.org/`.

[2] OpenFlow. `http://www.openflow.org`.

[3] NTT, in collaboration with Nicira Networks, succeeds in remote datacenter live migration, August 2011. `http://www.ntt.co.jp/news2011/1108e/110802a.html`.

[4] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: Virtual machine co-migration for the cloud. In *Proc. International Symposium on High Performance Distributed Computing (HPDC)*, 2011.

[5] A. Bhaskara, M. Charikar, E. Chlamtac, U. Feige, and A. Vijayaraghavan. Detecting high log-densities: An o($n^{1/4}$) approximation for densest k-subgraph. In *ACM Symposium on Theory of Computing (STOC)*, 2010.

[6] BigSwitch Networks. Perspectives: Networking needs a VMware. `http://www.bigswitch.com/wp/about-us/`.

[7] A. Botta, A. Dainotti, and A. Pescapè. Multi-protocol and multi-platform traffic generation and measurement. In *INFOCOM 2007 DEMO Session*, May 2007.

[8] R. Buyya and S. Venugopal. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proc. International Conference on Cloud Computing (CloudCom 2009)*, December 2009.

[9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[10] Cisco and VMWare. Virtual machine mobility with VMware VMotion and Cisco data center interconnect technologies, 2009.

[11] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proc. International Symposium on High Performance Distributed Computing (HPDC)*, 2011.

[12] D. Erickson et al. A demonstration of virtual machine mobility in an OpenFlow network, August 2008. Demo at *ACM SIGCOMM*.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *ACM SIGCOMM Computer Communications Review*, 38(3), 2008.

[14] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, August 2009. Demo at *ACM SIGCOMM*.

[15] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *Networked Systems Design and Implementation*, April 2010.

[16] E. Keller, J. Rexford, and J. van der Merwe. Seamless BGP Migration with Router Grafting. In *Proc. Networked Systems Design and Implementation (NSDI)*, 2010.

[17] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM HotNets Workshop*, pages 1–6, 2010.

[18] C.-C. Lin, M. Caesar, and J. van der Merwe. Towards interactive debugging for ISP networks. In *HotNets-VIII*, October 2009.

[19] J. Markoff. Open networking foundation pursues new standards. *The New York Times*, March 2011. See `http://nyti.ms/eK3CCK`.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communications Review*, 38(2):69–74, 2008.

[21] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. A. Corrêa, S. C. de Lucena, and M. F. Magalhães. Virtual routers as a service: The RouteFlow approach leveraging software-defined networks. In *International Conference on Future Internet Technologies*, June 2011.

[22] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control in enterprise networks. In *Workshop on Research on Enterprise Networking*, August 2009.

[23] P. S. Pisa, N. C. Fernandes, H. E. T. Carvalho, M. D. D. Moreira, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte. OpenFlow and Xen-based virtual network migration. In *The World Computer Congress – Network of the Future Conference*, 2010.

[24] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security*, November 2009.

[25] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Passive and Active Measurement Conference*, March 2012.

[26] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE Workshop*, March 2011.

[27] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: Live router migration as a network-management primitive. In *ACM SIGCOMM*, 2008.

[28] K. C. Webb, A. C. Snoeren, and K. Yocum. Topology switching for data center networks. In *Hot-ICE Workshop*, March 2011.

[29] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines. In *Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2011.

[30] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proceedings of Usenix Annual Technical Conference*, June 2011.

[31] Zynga. United States Security and Exchange Commission form S-1, 2011.